EXHIBIT 9

Maker Docs

Oracle Security Module (OSM) - Detailed Documentation

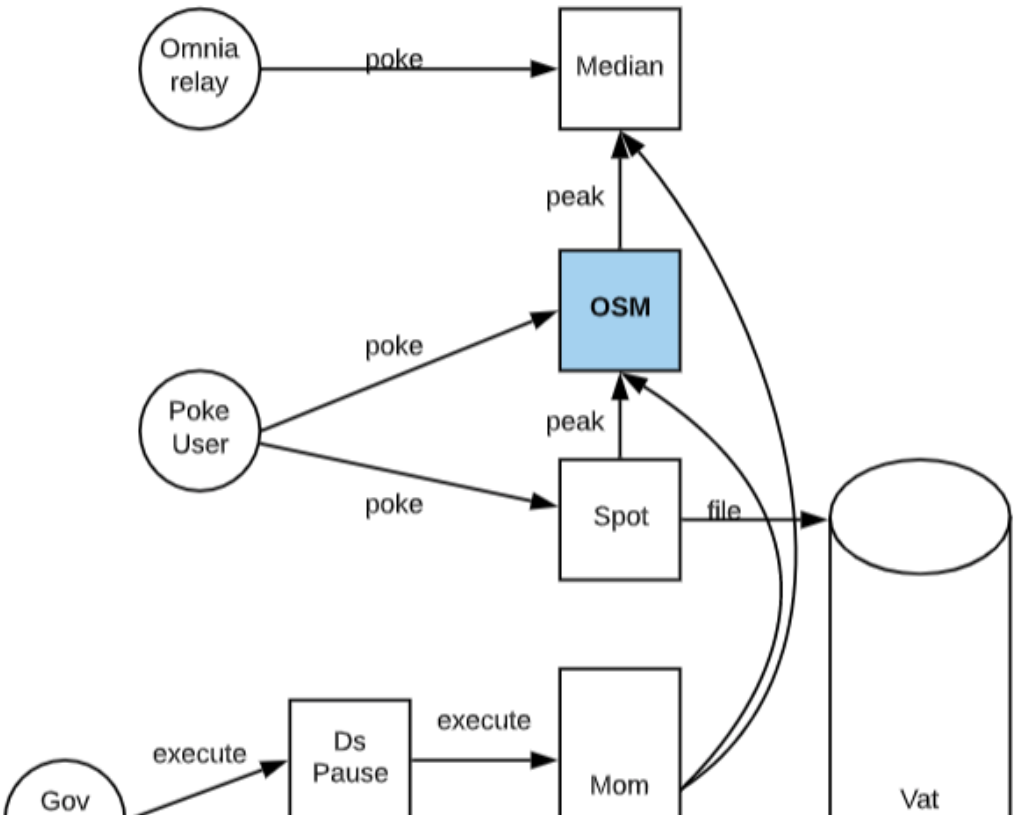
- **Contract Name:** OSM
- **Type/Category:** Oracles - Price Feed Module
- **Associated MCD System Diagram**
- **Contract Source**

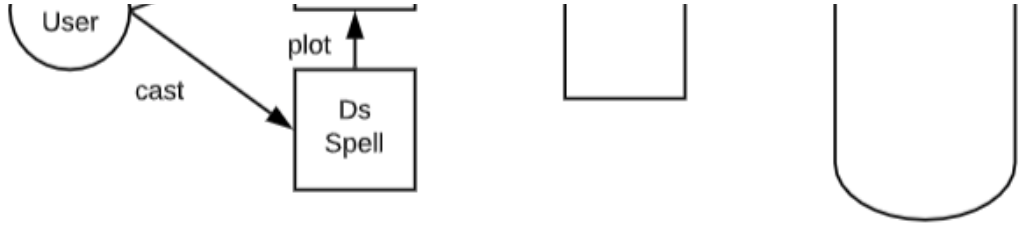
---

1. Introduction

Summary

The OSM (named via acronym from "Oracle Security Module") ensures that new price values propagated from the Oracles are not taken up by the system until a specified delay has passed. Values are read from a designated DSValue contract (or any contract that has the `read()` and `peek()` interfaces) via the `poke()` method; the `read()` and `peek()` methods will give the current value of the price feed, and other contracts must be whitelisted in order to call these. An OSM contract can only read from a single price feed, so in practice one OSM contract must be deployed per collateral type.





---

2. Contract Details - Glossary (OSM)

Storage Layout

- `stopped` : flag (`uint256`) that disables price feed updates if non-zero
- `src` : `address` of DSValue that the OSM will read from
- `ONE_HOUR` : 3600 seconds (`uint16(3600)`)
- `hop` : time delay between `poke` calls (`uint16`); defaults to `ONE_HOUR`
- `zzz` : time of last update (rounded down to nearest multiple of `hop`)
- `cur` : `Feed` struct that holds the current price value
- `nxt` : `Feed` struct that holds the next price value
- `bud` : mapping from `address` to `uint256`; whitelists feed readers

Public Methods

Administrative Methods

These functions can only be called by authorized addresses (i.e. addresses `usr` such that `wards[usr] == 1`).

- `rely` / `deny` : add or remove authorized users (via modifications to the `wards` mapping)
- `stop()` / `start()` : toggle whether price feed can be updated (by changing the value of `stopped`)
- `change(address)` : change data source for prices (by setting `src`)
- `step(uint16)` : change interval between price updates (by setting `hop`)
- `void()` : similar to `stop`, except it also sets `cur` and `nxt` to a `Feed` struct with zero values
- `kiss(address)` / `diss(address)` : add/remove authorized feed consumers (via modifications to the `buds` mapping)

Feed Reading Methods

These can only be called by whitelisted addresses (i.e. addresses `usr` such that `buds[usr] == 1`):

-

- `peek()` : returns the current feed value and a boolean indicating whether it is valid
- `peep()` : returns the next feed value (i.e. the one that will become the current value upon the next `poke()` call), and a boolean indicating whether it is valid
- `read()` : returns the current feed value; reverts if it was not set by some valid mechanism

Feed Updating Methods

- `poke()` : updates the current feed value and reads the next one

`Feed` struct: a struct with two `uint128` members, `val` and `has`. Used to store price feed data.

---

3. Key Mechanisms & Concepts

The central mechanism of the OSM is to periodically feed a delayed price into the MCD system for a particular collateral type. For this to work properly, an external actor must regularly call the `poke()` method to update the current price and read the next price. The contract tracks the time of the last call to `poke()` in the `zzz` variable (rounded down to the nearest multiple of `hop`; see Failure Modes for more discussion of this), and will not allow `poke()` to be called again until `block.timestamp` is at least `zzz+hop`. Values are read from a designated DSValue contract (its address is stored in `src`). The purpose of this delayed updating mechanism is to ensure that there is time to detect and react to an Oracle attack (e.g. setting a collateral's price to zero). Responses to this include calling `stop()` or `void()`, or triggering Emergency Shutdown.

Other contracts, if whitelisted, may inspect the `cur` value via the `peek()` and `read()` methods (`peek()` returns an additional boolean indicating whether the value has actually been set; `read()` reverts if the value has not been set). The `nxt` value may be inspected via `peep()`.

The contract uses a dual-tier authorization scheme: addresses mapped to 1 in `wards` may start and stop, set the `src`, call `void()`, and add new readers; addresses mapped to 1 in `buds` may call `peek()`, `peep()`, and `read()`.

---

4. Gotchas (Potential Sources of User Error)

Confusing `peek()` for `peep()` (or vice-versa)

The names of these methods differ by only a single character and in current linguistic usage, both "peek" and "peep" have essentially the same meaning. This makes it easy for a developer to confuse the two and call the wrong one. The effects of such an error are naturally context-dependent, but could e.g. completely invalidate the purpose of the OSM if the `peep()` is called where instead `peek()` should be used. A mnemonic to help distinguish them: "since 'k' comes before 'p' in the English alphabet, the value returned by `peek()` comes before the value returned by `peep()` in chronological order". Or: "`peek()` returns the **k**urrent value".

5. Failure Modes (Bounds on Operating Conditions & External Risk Factors)

`poke()` is not called promptly, allowing malicious prices to be swiftly uptaken

For several reasons, `poke()` is always callable as soon as `block.timestamp / hop` increments, regardless of when the last `poke()` call occurred (because `zzz` is rounded down to the nearest multiple of `hop`). This means the contract does not actually guarantee that a time interval of at least `hop` seconds has passed since the last `poke()` call before the next one; rather this is only (approximately) guaranteed if the last `poke()` call occurred shortly after the previous increase of `block.timestamp / hop`. Thus, a malicious price value can be acknowledged by the system in a time potentially much less than `hop`.

This was a deliberate design decision. The arguments that favoured it, roughly speaking, are:

- Providing a predictable time at which MKR holders should check for evidence of oracle attacks (in practice, `hop` is 1 hour, so checks must be performed at the top of the hour)
- Allowing all OSMs to be reliably poked at the same time in a single transaction

The fact that `poke` is public, and thus callable by anyone, helps mitigate concerns, though it does not eliminate them. For example, network congestion could prevent anyone from successfully calling `poke()` for a period of time. If an MKR holder observes that `poke` has not been promptly called, **the actions they can take include:**

1. Call `poke()` themselves and decide if the next value is malicious or not
2. Call `stop()` or `void()` (the former if only `nxt` is malicious; the latter if the malicious value is already in `cur`)
3. Trigger emergency shutdown (if the integrity of the overall system has already been compromised or if it is believed the rogue oracle(s) cannot be fixed in a reasonable length of time)

In the future, the contract's logic may be tweaked to further mitigate this (e.g. by **only** allowing `poke()` calls in a short time window each `hop` period).

Authorization Attacks and Misconfigurations

Various damaging actions can be taken by authorized individuals or contracts, either maliciously or accidentally:

- Revoking access of core contracts to the methods that read values, causing mayhem as prices fail to update
- Completely revoking all access to the contract
- Changing `src` to either a malicious contract or to something that lacks a `peek()` interface, causing transactions that `poke()` the affected OSM to revert

- Calling disruptive functions like `stop` and `void` inappropriately

The only solution to these issues is diligence and care regarding the `wards` of the OSM.

Maker Docs

Oracle Module

The Maker Protocol's Oracles

- **Module Name:** Oracle Module
- **Type/Category:** Oracles —> OSM.sol & Median.sol
- **Associated MCD System Diagram**
- **Contract Sources:**
  - **Median**
  - **OSM**

---

1. Introduction (Summary)

An oracle module is deployed for each collateral type, feeding it the price data for a corresponding collateral type to the `Vat`. The Oracle Module introduces the whitelisting of addresses, which allows them to broadcast price updates off-chain, which are then fed into a `median` before being pulled into the `OSM`. The `Spot` 'ter will then proceed to read from the `OSM` and will act as the liaison between the `oracles` and `dss`.

---

2. Module Details

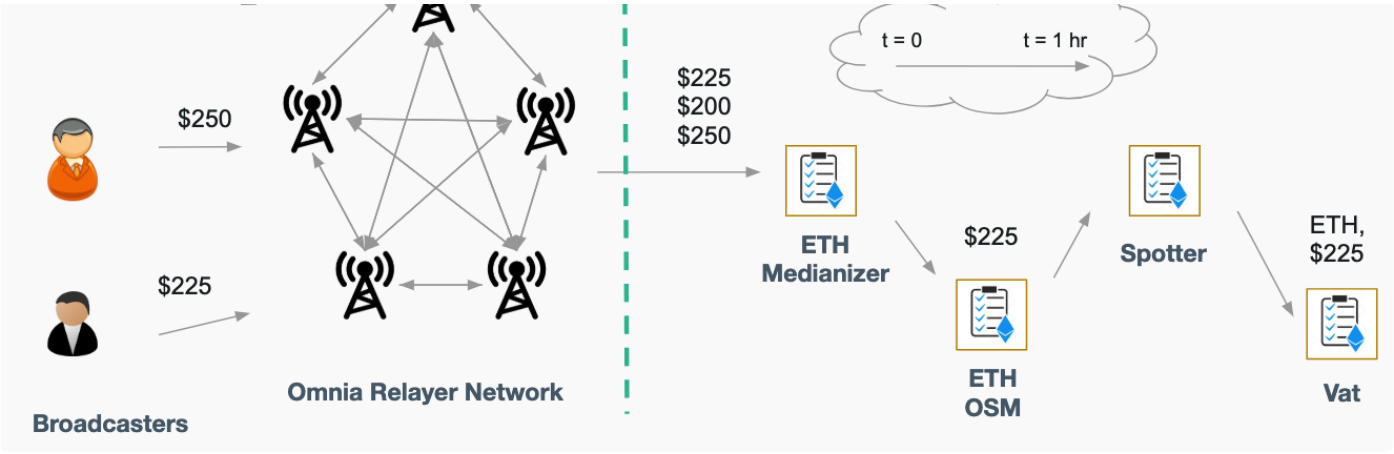
The Oracle Module has 2 core components consisting of the `Median` and `OSM` contracts.

Oracle Module Components Documentation

- **Median Documentation**
- **OSM Documentation**

---

3. Key Mechanism and Concepts



Interaction Diagram (Credit: MCD-101 Presentation, by Kenton Prescott)

Summary of the Oracle **Module Components**

- The `Median` provides Maker's trusted reference price. In short, it works by maintaining a whitelist of price feed contracts which are authorized to post price updates. Every time a new list of prices is received, the median of these is computed and used to update the stored value. The median has permissioning logic which is what enables the addition and removal of whitelisted price feed addresses that are controlled via governance. The permissioning logic allows governance to set other parameters that control the Median's behavior—for example, the `bar` parameter is the minimum number of prices necessary to accept a new median value.
- The `OSM` (named via acronym from "Oracle Security Module") ensures that new price values propagated from the Oracles are not taken up by the system until a specified delay has passed. Values are read from a designated DSValue contract (or any contract that implements the `read()` and `peek()` interface) via the `poke()` method; the `read()` and `peek()` methods will give the current value of the price feed, and other contracts must be whitelisted in order to call these. An OSM contract can only read from a single price feed, so in practice one OSM contract must be deployed per collateral type.

---

4. Gotchas (Potential sources of user error)

Relationship between the OSM and the Median:

- You can read straight from the median and in return, you would get a more real-time price. However, this depends on the cadence of updates (calls to poke).
- The OSM is similar but has a 1-hour price delay. It has the same process for reading (whitelist, auth, read and peek) as a median. The way the OSM works, is you cannot update it directly but you can `poke` it to go and read from something that also has the same structure (the `peek` method - in this case, its the median but you can set it to read from anything that conforms to the same interface).
- Whenever the OSM reads from a source, it queues the value that it reads for the following hour or following `hop` property, which is set to 1 hour (but can be anything). When it is `poke`'d, it reads the value of the median and it will save the value. Then the previous value becomes that, so it is always off by an hour. After an hour passes, when `poke`d, the value that it saved becomes the current value and

whatever value is in the median becomes the future value for the next hour.

- `spot` - if you poke it with an ilk (ex: ETH) it will read form the OSM and if the price is valid, it updates.

Relationship to the `Spot` 'ter:

- In relation to the `Spot` the oracle module handles how market prices are recorded on the blockchain. The `Spot` ter operates as the interface contract, which external actors can use to retrieve the current market price from the Oracle module for the specified collateral type. The `Vat` in turn reads the market price from the `spot` ter.

---

5. Failure Modes (Bounds on Operating Conditions & External Risk Factors)

- `Median` - there is currently no way to turn off the oracle (failure or returns false) if all the oracles come together and sign a price of zero. This would result in the price being invalid and would return false on `peek`, telling us to not trust the value.
- `OSM`
  - `poke()` is not called promptly, allowing malicious prices to be swiftly uptaken.
  - Authorization Attacks and Misconfigurations.
  - Read more here.

Oracle Security Module (OSM) - Detailed Documentation

- **Contract Name:** OSM
- **Type/Category:** Oracles - Price Feed Module
- **Associated MCD System Diagram**
- **Contract Source**

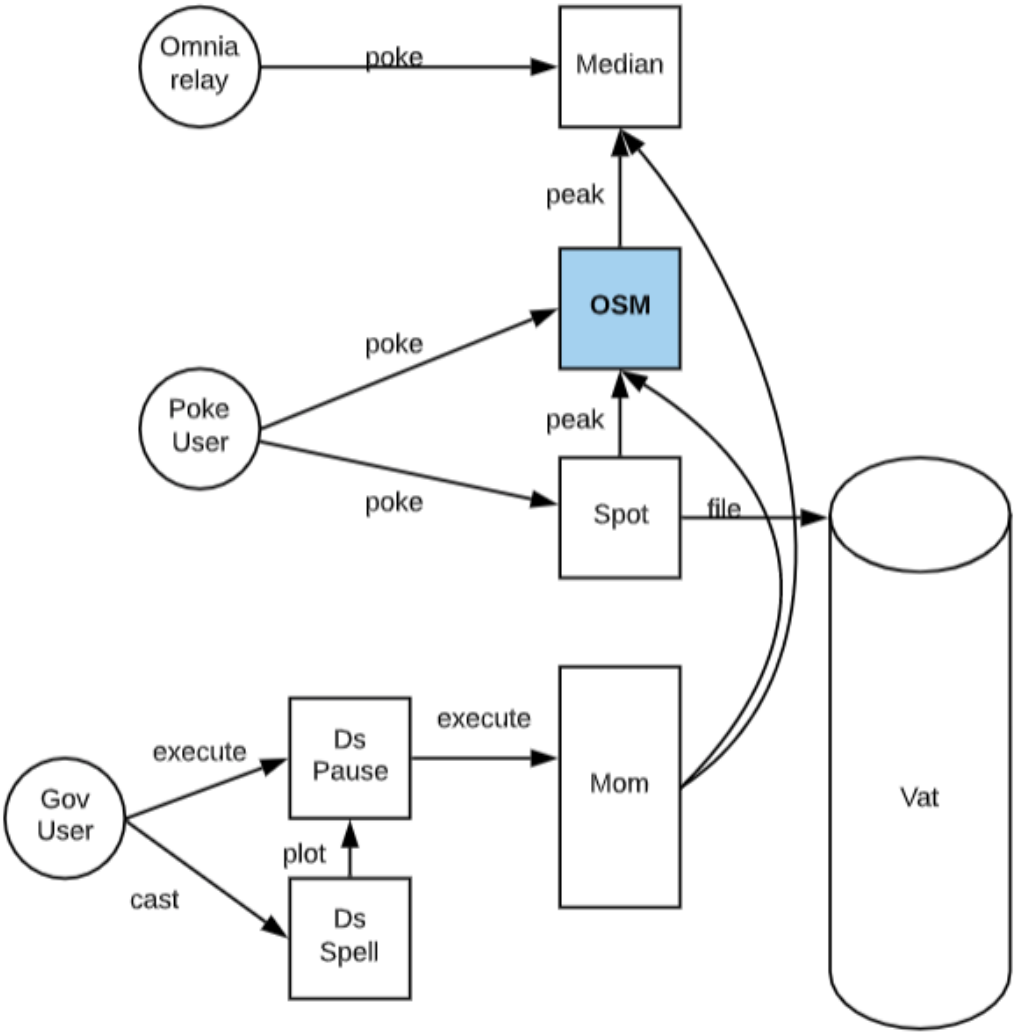
---

1. Introduction

Summary

The OSM (named via acronym from "Oracle Security Module") ensures that new price values propagated from the Oracles are not taken up by the system until a specified delay has passed. Values are read from a designated DSValue contract (or any contract that has the `read()` and `peek()` interfaces) via the

`poke()` method; the `read()` and `peek()` methods will give the current value of the price feed, and other contracts must be whitelisted in order to call these. An OSM contract can only read from a single price feed, so in practice one OSM contract must be deployed per collateral type.



---

2. Contract Details - Glossary (OSM)

Storage Layout

- `stopped` : flag (`uint256`) that disables price feed updates if non-zero
- `src` : `address` of DSValue that the OSM will read from
- `ONE_HOUR` : 3600 seconds (`uint16(3600)`)
- `hop` : time delay between `poke` calls (`uint16`); defaults to `ONE_HOUR`

- `zzz` : time of last update (rounded down to nearest multiple of `hop`)

- `cur` : `Feed` struct that holds the current price value
- `nxt` : `Feed` struct that holds the next price value
- `bud` : mapping from `address` to `uint256`; whitelists feed readers

Public Methods

Administrative Methods

These functions can only be called by authorized addresses (i.e. addresses `usr` such that `wards[usr] == 1`).

- `rely` / `deny` : add or remove authorized users (via modifications to the `wards` mapping)
- `stop()` / `start()` : toggle whether price feed can be updated (by changing the value of `stopped`)
- `change(address)` : change data source for prices (by setting `src`)
- `step(uint16)` : change interval between price updates (by setting `hop`)
- `void()` : similar to `stop`, except it also sets `cur` and `nxt` to a `Feed` struct with zero values
- `kiss(address)` / `diss(address)` : add/remove authorized feed consumers (via modifications to the `buds` mapping)

Feed Reading Methods

These can only be called by whitelisted addresses (i.e. addresses `usr` such that `buds[usr] == 1`):

- `peek()` : returns the current feed value and a boolean indicating whether it is valid
- `peep()` : returns the next feed value (i.e. the one that will become the current value upon the next `poke()` call), and a boolean indicating whether it is valid
- `read()` : returns the current feed value; reverts if it was not set by some valid mechanism

Feed Updating Methods

- `poke()` : updates the current feed value and reads the next one

`Feed` struct: a struct with two `uint128` members, `val` and `has`. Used to store price feed data.

---

3. Key Mechanisms & Concepts

The central mechanism of the OSM is to periodically feed a delayed price into the MCD system for a particular collateral type. For this to work properly, an external actor must regularly call the `poke()` method to update the current price and read the next price. The contract tracks the time of the last call to `poke()` in the `zzz` variable (rounded down to the nearest multiple of `hop`; see Failure Modes for more discussion of this), and will not allow `poke()` to be called again until `block.timestamp` is at least `zzz+hop`.

Values are read from a designated DSValue contract (its address is stored in `src`). The purpose of this delayed updating mechanism is to ensure that there is time to detect and react to an Oracle attack (e.g. setting a collateral's price to zero). Responses to this include calling `stop()` or `void()`, or triggering Emergency Shutdown.

Other contracts, if whitelisted, may inspect the `cur` value via the `peek()` and `read()` methods (`peek()` returns an additional boolean indicating whether the value has actually been set; `read()` reverts if the value has not been set). The `nxt` value may be inspected via `peep()`.

The contract uses a dual-tier authorization scheme: addresses mapped to 1 in `wards` may start and stop, set the `src`, call `void()`, and add new readers; addresses mapped to 1 in `buds` may call `peek()`, `peep()`, and `read()`.

---

4. Gotchas (Potential Sources of User Error)

Confusing `peek()` for `peep()` (or vice-versa)

The names of these methods differ by only a single character and in current linguistic usage, both "peek" and "peep" have essentially the same meaning. This makes it easy for a developer to confuse the two and call the wrong one. The effects of such an error are naturally context-dependent, but could e.g. completely invalidate the purpose of the OSM if the `peep()` is called where instead `peek()` should be used. A mnemonic to help distinguish them: "since 'k' comes before 'p' in the English alphabet, the value returned by `peek()` comes before the value returned by `peep()` in chronological order". Or: "`peek()` returns the **k**urrent value".

---

5. Failure Modes (Bounds on Operating Conditions & External Risk Factors)

`poke()` is not called promptly, allowing malicious prices to be swiftly uptaken

For several reasons, `poke()` is always callable as soon as `block.timestamp / hop` increments, regardless of when the last `poke()` call occurred (because `zzz` is rounded down to the nearest multiple of `hop`). This means the contract does not actually guarantee that a time interval of at least `hop` seconds has passed since the last `poke()` call before the next one; rather this is only (approximately) guaranteed if the last `poke()` call occurred shortly after the previous increase of `block.timestamp / hop`. Thus, a malicious price value can be acknowledged by the system in a time potentially much less than `hop`.

This was a deliberate design decision. The arguments that favoured it, roughly speaking, are:

- Providing a predictable time at which MKR holders should check for evidence of oracle attacks (in practice, `hop` is 1 hour, so checks must be performed at the top of the hour)
- Allowing all OSMs to be reliably poked at the same time in a single transaction

The fact that `poke` is public, and thus callable by anyone, helps mitigate concerns, though it does not eliminate them. For example, network congestion could prevent anyone from successfully calling `poke()` for a period of time. If an MKR holder observes that `poke` has not been promptly called, **the actions they can take include:**

1. Call `poke()` themselves and decide if the next value is malicious or not
2. Call `stop()` or `void()` (the former if only `nxt` is malicious; the latter if the malicious value is already in `cur`)
3. Trigger emergency shutdown (if the integrity of the overall system has already been compromised or if it is believed the rogue oracle(s) cannot be fixed in a reasonable length of time)

In the future, the contract's logic may be tweaked to further mitigate this (e.g. by **only** allowing `poke()` calls in a short time window each `hop` period).

Authorization Attacks and Misconfigurations

Various damaging actions can be taken by authorized individuals or contracts, either maliciously or accidentally:

- Revoking access of core contracts to the methods that read values, causing mayhem as prices fail to update
- Completely revoking all access to the contract
- Changing `src` to either a malicious contract or to something that lacks a `peek()` interface, causing transactions that `poke()` the affected OSM to revert
- Calling disruptive functions like `stop` and `void` inappropriately

The only solution to these issues is diligence and care regarding the `wards` of the OSM.

Median - Detailed Documentation

The Maker Protocol's trusted reference price

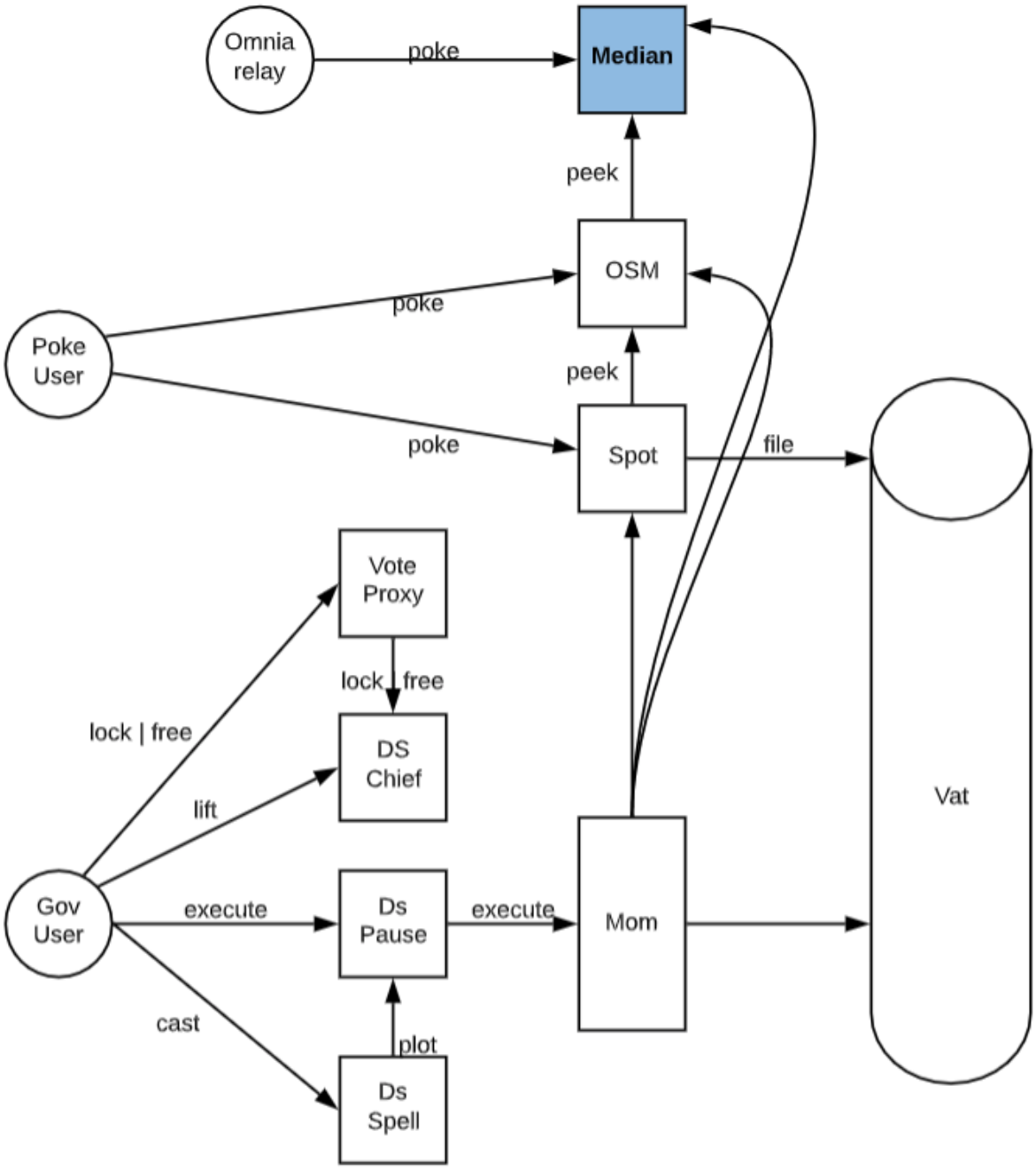
- **Contract Name:** median.sol
- **Type/Category:** Oracles Module
- **Associated MCD System Diagram**
- **Contract Source**

---

1. Introduction (Summary)

The median provides Maker's trusted reference price. In short, it works by maintaining a whitelist of price feed contracts which are authorized to post price updates. Every time a new list of prices is received, the median of these is computed and used to update the stored value. The median has permissioning logic which is what enables the addition and removal of whitelisted price feed addresses that are controlled via governance. The permissioning logic allows governance to set other parameters that control the Median's behavior—for example, the `bar` parameter is the minimum number of prices necessary to accept a new median value.

A High-level overview diagram of the components that involve and interact with the median:



Note: All arrows without labels are governance calls.

2. Contract Details

Median (Glossary)

Key Functionalities (as defined in the smart contract)

- `read` - Gets a non-zero price or fails.
- `peek` - Gets the price and validity.
- `poke` - Updates price from whitelisted providers.
- `lift` - Adds an address to the writers whitelist.
- `drop` - Removes an address from the writers whitelist.
- `setBar` - Sets the `bar`.
- `kiss` - Adds an address to the reader's whitelist.
- `diss` - Removes an address from the readers whitelist.

Note: `read` returns the `value` or fails if it's invalid & `peek` gives back the `value` and if the `value` is valid or not.

Other

- `wards(usr: address)` - Auth mechanisms.
- `orcl(usr: address)` - `val` writers whitelist / signers of the prices (whitelisted via governance / the authorized parties).
- `bud(usr: address)` - `val` readers whitelist.
- `val` - the price (private) must be read with `read()` or `peek()`
- `age` - the Block timestamp of last price `val` update.
- `wat` - the price oracles type (ex: ETHUSD) / tells us what the type of asset is.
- `bar` - the Minimum writers quorum for `poke` / min number of valid messages you need to have to update the price.

---

3. Key Mechanisms & Concepts

As mentioned above, the `median` is the smart contract that provides Maker's trusted reference price. Authorization **(auth)** is a key component included in the mechanism of this contract and its interactions. For example, the price (`val`) is intentionally kept not public because the intention is to only read it from the two functions `read` and `peek`, which are whitelisted. This means that you need to be authorized, which is completed through the `bud`. The `bud` is modified to get whitelisted authorities to read it on-chain (permissioned), whereas, everything of off-chain is public.

The `poke` method is not under any kind of `auth`. This means that anybody can call it. This was designed for the purpose of getting Keepers to call this function and interact with Auctions. The only way to modify its

state is if you call it and send it valid data. For example, let's say this oracle needs 15 different sources. This means that we would need it to send 15 different signatures. It will then proceed to go through each of them and validate that whoever sent the the data has been `auth`'d to do so. In the case of it being an authorized oracle, it will check if it signed the message with a timestamp that is greater than the last one. This is done for the purpose of ensuring that it is not a stale message. The next step is to check for order values, this requires that you send everything in an array that is formatted in ascending order. If not sent in the correct order (ascending), the median is not calculated correctly. This is because if you assume the prices are ordered, it would just grab the middle value which may not be sufficient or work. In order to check for uniqueness, we have implemented the use of a `bloom` filter. In short, a bloom filter is a data structure designed to tell us, rapidly and memory-efficiently, whether an element is present in a set. This use of the bloom filter helps with optimization. In order to whitelist signers, the first two characters of their addresses (the first `byte`) have to be unique. For example, let's say that you have 15 different price signers, none of the first two characters of their addresses can be the same. This helps to filter that all 15 signers are different.

Next, there are `lift` functions. These functions tell us who can sign messages. Multiple messages can be sent or it can just be one but they are put into the authorized oracle). However, there is currently nothing preventing someone from `lift`'ing two prices signers that start with the same address. This is something for example, that governance needs to be aware of (see an example of what a governance proposal would look like in this case in the **Gotchas** section).

Due to the mechanism design of how the oracles work, the **quorum** has to be an odd number. If it is an even number, it will not work. This was designed as an optimization (`val = uint128(val_[val_.length >> 1]);`); this code snippet outlines how it works, which is by taking the array of values (all the prices that each of the prices signers reported, ordered from 200-215) and then grabbing the one in the middle. This is done by taking the length of the array (15) and shifting it to the right by 1 (which is the same as dividing by 2). This ends up being 7.5 and then the EVM floors it to 7. If we were to accept even numbers this would be less efficient. This presents the issue that you should have a defined balance between how many you require and how many signers you actually have. For example, let's say the oracle needs 15 signatures, you need at least 17-18 signers because if you require 15 and you only have 15 and one of them goes down, you have no way of modifying the price, so you should always have a bit more. However, you should not have too many, as it could compromise the operation.

---

4. Gotchas

Emergency Oracles

- They can shutdown the price feed but cannot bring it back up. Bringing the price feed back up requires governance to step in.

Price Freeze

- If you void the oracles Ethereum module, the idea is that you cannot interact with any Vault that depends on that ilk.
    - **Example:** ETHUSD shutdown (can still add collateral and pay back debt - increases safety) but you

cannot do anything that increases risk (decreases safety - remove collateral, generate dai, etc.) because the system would not know if you would be undercollateralized.

Oracles Require a lot of Upkeep

- They need to keep all relayers functioning.
- The community would need to self-police (by looking at each price signer, etc.) if any of them needs to be replaced. They would need to make sure they are constantly being called every hour (for every hour, a transaction gets sent to the OSM, which means that a few transactions have already been sent to the median to update it as well. In addition, there would need to be a transaction sent to the `spotter`, as DSS operates in a pool-type method (doesn't update the system/write to it, you tell it to read it from the OSM).

There is nothing preventing from `lift`'ing two prices signers that start with the same address

- The only thing that this prevents is that you cannot have more than 256 oracles but we don't expect to ever have that many, so it is a hard limit. However, Governance needs to be sure that whoever they are voting in anyone that they have already voting in before with the same two first characters.
- An example of what a governance proposal would look like in this case:
  - We are adding a new oracle and are proposing (the Foundation) a list of signers (that have been used in the past) and we already have an oracle but want to add someone new (e.g. Dharma or dydx). We would say that they want to be price signers, so these are their addresses and we want to lift those two addresses. They would vote for that, and we would need to keep a list of the already existing addresses and they would need to create an address that doesn't conflict with the existing ones.

---

5. Failure Modes (Bounds on Operating Conditions & External Risk Factors)

- By design, there is currently no way right now to turn off the oracle (failure or returns false) if all the oracles come together and sign a price of zero. This would result in the price being invalid and would return false on `peek`, telling us to not trust the value.
  - We are currently researching (Oracles ETH module) that would invalidate the price but there is no way to do this in the median today. This is due to the separation of concerns that DSS does not read directly from median, it reads from the OSM, but this may end up changing.